
**Division of Informatics, University of Edinburgh**

**Centre for Intelligent Systems and their Applications**

# Logic-based Program Synthesis via Program Extraction

by

Ewen Denney

**Informatics Research Report EDI-INF-RR-0142**

**Division of Informatics** **March 2002**
http://www.informatics.ed.ac.uk/

# Logic-based Program Synthesis via Program Extraction

Ewen Denney

Informatics Research Report EDI-INF-RR-0142

DIVISION *of* INFORMATICS
Centre for Intelligent Systems and their Applications

March 2002

appears in AAAI 2002 Spring Symposium, Stanford, USA

**Abstract :**

This paper outlines the author's experiences using the methodology of program extraction within a proof assistant-based on constructive type theory. We discuss the feasibility of the methodology and tool support, and suggest some directions for future research.

**Keywords** : program synthesis, Java Card, type theory, abstraction

# Logic-based Program Synthesis via Program Extraction

POSITION PAPER

**Ewen Denney**
Division of Informatics
University of Edinburgh
Scotland
ewd@dai.ed.ac.uk

**Abstract**

This paper outlines the author's experiences using the methodology of program extraction within a proof assistant based on constructive type theory. We discuss the feasibility of the methodology and tool support, and suggest some directions for future research.

## Introduction

The author's interest in this area originated from an industrial collaboration between his research group, who were interested in formalising aspects of the Java Card virtual machine, and a company interested in achieving formal certification for their Java Card products.

Our group had already formalised a Java Card framework at the bytecode level, and a specification of a particular optimisation of interest, which had been used to verify its properties. The specification in question was, roughly speaking, of the form

$$\forall x : \texttt{Class\_file} \,.\, \exists y : \texttt{CAP\_file} \,.\, correct(x, y)$$

This is the traditional form of a synthesis problem so we decided to try and prove the theorem constructively (in Coq) and hence develop an actual optimisation algorithm by extraction.

The proof (of course) turned out to be far harder than we had anticipated. A program was finally obtained (Denney 2001), but even though we did expect the effort involved to be prohibitive, we were surprised that there appeared to be little in the literature by way of case studies, techniques, or tool support, which compounded the difficulty.

Constructive proof assistants, such as Coq, are frequently 'sold' as being a basis for program extraction, yet the embarrassing truth is that they are rarely used for this: why?

We address this question by looking at a number of issues concerned with the extraction methodology, and suggest several directions in which it could be improved.

## Discussion

We discuss four aspects of the extraction methodology: its logical basis, tactical support, the nature of the code produced, and its part in the software process.



### The Logical Basis of Synthesis

Some people are put off the whole synthesis paradigm because of its associations with constructive logic. There are a number of misconceptions here. Firstly, people tend to be more comfortable with classical logic, and see constructive logic, or *constructivism*, as a philosophical movement (more correctly, *intuitionism*), with little to do with computing science.

In actual fact, most synthesis-oriented reasoning is classical. Typically, *verification* can be done perfectly well in classical logic. Indeed, a large part of synthesis consists of verifying properties of components, so there is no need for an entirely constructive framework. (For example, so long as a witness $t$ to $\exists x.P$ is supplied, $P[t/x]$ can be proven classically). This then raises the question of how best to combine classical and constructive reasoning techniques.

Another problem is that existing corpora of proofs are typically in classical logics. Also, many decision procedures and familiar proof techniques such as resolution only work in classical logic (though resolution is constructive for Horn clauses). Some other questions to address are:

- what techniques are best for mining classical corpora? *e.g.* proof transformations?
- which existing tactics are constructive?
- when is mixing classical and constructive reasoning safe?

Some of these issues have been discussed before (*e.g.* see (Caldwell & Underwood 1996)), but although it is well-known that there are various techniques for getting computational content from classical proofs, these have not been seriously exploited.

An exception is the recent work of Schwichtenberg (Schwichtenberg 1999), who has developed refined techniques for extracting content from a subset of classical proofs, and implemented this in the Minlog system.

### Proof Techniques

Typically, the tactics which are needed for synthesis have a slightly different emphasis than those used in 'standard' theorem proving. The main problem is what might be described as "constructing objects from other objects".

There are two problems here. First, how do we decide which objects are relevant (given a library of programs and lemmas) and, then, how do we join them together?

For example, suppose we want to construct a program which satisfies the specification $\phi \to \chi$ (meaning, if the input satisfies $\phi$ then the output satisfies $\chi$) given programs which satisfy $\phi \to \psi$ and $\psi' \to \chi$, where $\psi$ and $\psi'$ are 'similar', in some sense. How do we characterise and recognise this similarity, and how do we combine the programs?

A common situation is that specifications and proofs are often complicated by well-formedness constraints (which are often decidable), but the proof is best explained at a certain level of abstraction, (*e.g.* once we abstract from well-formedness constraints, some proofs can be viewed as equality reasoning; others are simple propositional reasoning.)

In fact, these two themes — abstraction and relevance — are related. *Gazing* (Barker-Plummer 1992) is an abstraction-based technique which aims to select relevant lemmas for use in proofs. To apply this technique here, we would replace "similar" above with "have a common abstraction".

However, this technique has never been applied in a constructive setting and, in particular, to program synthesis. The extension of (Barker-Plummer 1992) to a higher-order constructive setting is not obvious, but looks like a promising way of applying component-based design in a constructive setting.

As an example of the application of gazing in a constructive context, consider the problem of using a component which is the curried equivalent of what would otherwise match a goal. An appropriate abstraction, however, will not distinguish between a term and its currying, so the proof can proceed. Later, the proof can be patched to account for such structural rearrangements.

### Architecture management

Another problem with the extraction paradigm is the confusing nature of the code that results. The problem, basically, is that extracted programs tend not to follow good principles of software design such as modularity, data abstraction, and sufficient documentation. Clearly, we can not expect a badly structured proof to give rise to a well-structured program. However, we should not expect things to get worse!

It could be argued that we should think of the proof as the 'source' and extraction as a form of compilation, and we should not worry about the form of the code. Nevertheless, in practice, we would prefer good code to bad, and the situation could be improved in several ways. In the case of Coq, the extraction mechanism should be able to at least reflect the logical structure of the proof in the physical layout of the functions. More generally, what we would like is that extraction does not just produce a program, but a program with *metadata* derived from the proof. We could also annotate the proof to supply information which cannot be derived automatically.

A related problem is that of displaying the proof comprehensibly. Coq (v6) has a natural language rendering mechanism, but this does just not work with constructive proofs. It could be extended, also incorporating tactics to give a hierarchical presentation of the proof (rather than the low-level steps which the tactics give rise to).

We can envisage two possible approaches to these problems. One is to represent the proof in XML and use its features to annotate the proof. Another is to stick with the proofs-as-programs paradigm but extend it to account for hierarchical proofs, on the logical side, and analogously hierarchical lambda terms, on the type-theory side.

### Software Process

Research on program synthesis has often been tempered by an isolationist attitude that synthesis is separate from the rest of the software process. A consequence of viewing it as being purely in the logical world is that little thought has been given to methodological concerns, such as the nature of the resulting code.

This has led to synthesis and extraction, in particular, as being viewed as some kind of 'Magic Button'. This leads to disillusionment. In practice, of course, synthesis is just one part of the software process, and depends upon other stages of development. As a methodology, extraction ranges from direct programming in a high-level language (effectively a form of automated data refinement), to full synthesis.

One of the surprises of (Denney 2001), was the need for testing the extracted programs — when the specification is not tight enough, or to validate the specifications generally; also, feedback is needed to see if the underlying data format is appropriate.

## Conclusion

It has been claimed that program extraction is an infeasible approach to formal software development. At the very least, I would say that reports of its demise are premature. There are so many, apparently untackled, problems that more research is needed to determine how feasible extraction could be.

The author is involved in a number of projects which address some of the issues raised here.

## Acknowledgements

This work has been funded by EPSRC grant GR/M45030.

## References

Barker-Plummer, D. 1992. Gazing: An approach to the problem of definition and lemma use. *J. Automated Reasoning* 8:311–344.

Caldwell, J., and Underwood, J. 1996. Classical tools for constructive proof search. In Galmiche, D., ed., *Proceedings of the CADE-13 Workshop on Proof search in type-theoretic languages*. Rutgers N.J.

Denney, E. 2001. The Synthesis of a Java Card Tokenisation Algorithm. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering, San Diego, USA*.

Schwichtenberg, H. 1999. Refined Program Extraction from Classical Proofs: Some Case Studies. In Broy, M., ed., *Lecture Notes of Marktoberdorf International Summer School*.